Technical Report

REMPLI Discreet Event Simulation System

Luís Marques

Filipe Pacheco

HURRAY-TR-070903

Version: 0

Date: 09-20-2007

REMPLI Discreet Event Simulation System

Luís Marques, Filipe Pacheco

IPP-HURRAY!

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8340509

E-mail: {lmarques, ffp}@isep.ipp.pt

http://www.hurray.isep.ipp.pt

Abstract

This document makes a brief introduction on the REMPLI Discreet Event Simulation system used to test the REMPLI Transport Layer.

1 Introduction

This document presents the design choices on the simulation mechanism used to test the Tansport Layer implementation in the REMPLI project [www.rempli.org].

2 REMPLI Event Simulation

2.1 Simulation Advantages

One of the primary advantages of simulators is that they are able to provide users with practical feedback when designing real world systems. This allows the designer to determine the correctness and efficiency of a design before the system is actually constructed. Consequently, the user may explore the advantages and disadvantages of alternative designs without actually physically building the systems. By investigating the effects of specific design decisions during the design phase rather than the construction phase, the overall cost of building the system diminishes significantly. By mimicking the behaviour of the designs, the circuit simulator is able to provide the designer with information pertaining to the correctness and efficiency of alternate designs. After carefully weighing the ramifications of each design, the best circuit may then be fabricated.

Another benefit of simulators is that they permit system designers to study a problem at several different levels of abstraction. By approaching a system at a higher level of abstraction, the designer is better able to understand the behaviours and interactions of all the high level components within the system and is therefore better equipped to counteract the complexity of the overall system. This complexity may simply overwhelm the designer if the problem had been approached from a lower level. As the designer better understands the operation of the higher level components through the use of the simulator, the lower level components may then be designed and subsequently simulated for verification and performance evaluation. The entire system may be built based upon this "top-down" technique. This approach is often referred to as hierarchical decomposition and is essential in any design tool and simulator which deals with the construction of complex systems. Working at a higher level abstraction also facilitates rapid prototyping in which preliminary systems are designed quickly for the purpose of studying the feasibility and practicality of the high-level design.

Thirdly, simulators can be used as an effective means for demonstrating concepts, and to analyse system behaviour whenever a fully-developed system is not available. This is particularly true of simulators that make intelligent use of computer graphics and animation. Such simulators dynamically show the behaviour and relationship of all the simulated system's components, thereby providing the user with a meaningful understanding of the system's nature. [Donald Craig, "Extensible Hierarchical Object-Oriented Logic Simulation with an Adaptable Graphical User Interface", MSc Thesis, Memorial University of Newfoundland]

2.2 Objectives

The main objective of the simulator is to accurately emulate the behaviour of the Power-Line physical medium and of the proposed layer protocols, allowing for a realistic analysis of them while the

system is not completely developed. The gathered information could also be used to improve or propose new protocols which would reduce developing costs.

Other objective is that the simulator should be modular allowing easy integration and swapping of other protocol layers. That would permit that different implementations of the same protocol layer coexist in the simulator and that the modules could be easy swapped or stacked allowing for an even more exhaustive and faster analysis of the network and different protocol proposals.

One other objective is to be able to use real protocol implementation code instead of just simulating code. That would allow for faster integration of protocol changes in the developing protocols resulting in time and costs saving.

2.3 Network Simulator 2 Tool

NS2 is an open-source simulation tool that runs on Linux. It is a discreet event simulator targeted at networking research and provides substantial support for simulation of routing, multicast protocols and IP protocols, such as UDP, TCP, RTP and SRM over wired and wireless (local and satellite) networks. It has many advantages that make it a useful tool, such as support for multiple protocols and the capability of graphically detailing network traffic. Additionally, NS2 supports several algorithms in routing and queuing. ["Network Simulator 2 - NS-2", Web Site: http://www.isi.edu/nsnam/ns/]

NS2 started as a variant of the REAL network simulator in 1989. REAL is a network simulator originally intended for studying the dynamic behaviour of flow and congestion control schemes in packet-switched data networks.

Currently NS2 development by VINT group is supported through researchers from Defense Advanced Research Projects Agency (DARPA) and National Science Foundation (NSF), both in collaboration with other researchers. NS2 is available on several platforms such as FreeBSD, Linux, SunOS and Solaris. NS2 is said to build and run under Windows, but it is not supported.

Simple scenarios can be run on any "reasonable" machine; however, very large scenarios benefit from large amounts of memory.

NS2 is built using a mix of C++ and TCL scripting. While the core of NS2 is made in fast C++ code, the protocol configuration is made in slower TCL script. This allows an easier and faster changing of parameters, and development of new protocol profiles without the need of recompiling the source code. Simulations are also configured in TCL.

2.3.1 Simulating with NS-2

For a simulation, we need to define all the network nodes (workstations and network devices), the connections between the nodes, the agents (protocols, layers, etc) in each node, and the applications (or services) that are run by the agents. We also need to configure the scheduling of network events, like the failure of a link at some instant of time or the starting of a traffic generator application.

It is also needed to define the wanted output variables – there are some tracing functions already built that generate tracing files for the graph application (that graphically shows the bandwidth usage of the system) and for the network animator (were we can see graphically what happens to the packets in hour network). For running the simulation we run NS giving the configuration TCL file as argument. It then simulates the network and gives the output we wanted.

2.3.2 NS2 adequate functionalities

- NS2 is modular and allows the implementation of new modules using C++, which would allow for the particular implementation of the PLC network devices behaviour as well as easy swapping of different protocol layers.

- Provides some graphical analysis tools which allow faster and more pleasant way of analysing simulation results. The user can replay simulations forwards or backwards and even increase simulation speed.
- Using simpler TCL files for configurations allows for faster changing of simulation parameters even by non-authors of the simulation modules, and requires no recompilation of the sources.

2.3.3 NS2 inadequate functionalities

- NS2 documentation is not accurate and requires the developer to spend much time guessing "how to do what". It seams to be "blindly" developed by many researchers, because many third party modules do not work in latest versions due to changes in NS2 core – this could also explain the inaccuracy of the documentation.
- NS2 learning curve is extremely steep, and although there are some tutorials to help get started, the knowledge required for implementing new modules and protocols will only come with actual, intensive and daily work usage.
- It is actually very hard to embed "foreign" code into NS2 modules which would disallow or dissuade the use of real implementation code.

2.4 OMNeT++ Tool

OMNeT++ is an object-oriented modular discrete event simulator. The name itself stands for Objective Modular Network Testbed in C++. Its primary application area is the simulation of communication networks, but because of its generic and flexible architecture, is successfully used in other areas like the simulation of complex IT systems, queuing networks or hardware architectures as well. [A. Varga. OMNeT++ Discrete Event Simulation System. v2.3, 2004. Web Site: http://www.omnetpp.org/]

OMNeT++ provides a component architecture for models. Components (modules) are programmed in C++, and then assembled into larger components and models using a high-level language (NED). The models are reusable, which is convenient for speeding up simulation development. OMNeT++ has an extensive GUI support, and due to its modular architecture, the simulation kernel (and models) can be embedded easily into applications. Although OMNeT++ is not a network simulator itself, it is currently gaining widespread popularity as a network simulation platform in the scientific community as well as in industrial settings, and building up a large user community.

OMNeT++ offers two modelling approaches:

- finite state machines, through the use of handleMessage() routine, which is called for each event;
- concurrent processes, through the use of activity(), which runs as a co-routine.

OMNeT++ can be used for:

- traffic modelling of telecommunication networks;
- protocol modelling;
- modelling queuing networks;
- modelling multiprocessors and other distributed hardware systems;
- validating hardware architectures;
- evaluating performance aspects of complex software systems.

2.4.1 Simulating with OMNET++

According to the modelling philosophy OMNeT++ implements, a system consists of a number of entities (modules) communicating by exchanging messages; modules can be nested, that is, several modules can be grouped together to form a bigger unit (a compound module). One of the first steps on building a simulation with OMNeT++ is to identify the components in one system model and their communication paths (connexions).

Then if any of those components is somewhat complex to implement totally with just one OMNeT++ module but can be divided into more specialized subcomponents then the optimal choice would be doing that division and to implement each sub-component as one sub-module (for example, in a computer network simulation were there are producer hosts and consumer hosts, the producer could be implemented as a module with the sub-modules "traffic generator" – to generate data to be sent to a consumer – "address chooser" – to choose a consumer address and encapsulate it in the data - and "checksum" – to calculate a checksum and to send the data to the network)

OMNeT++ allows for hierarchies of any depth but there is one drawback on doing this specialization: the simulation will have much more sub-modules and connexions which causes the simulation to run slower as there are much more messages going from one module to another as well as much more memory used. We cannot have the better of two worlds – simplicity and efficiency – so the level of simplicity should be chosen carefully.

OMNeT++ provides a graphical tool (GNED) to aid in the task of creating modules, which are defined in .NED files. Those files are readable and changeable in any text editor as they consist of plain text.

After all the modules and sub-modules are defined it is needed to describe the active components of the model as concurrent processes using C++. This is what will actually simulate the behavior of the target system. For example, in a switched network simulation, the module which implements the switch behavior would likely be waiting for incoming messages and when it receives a message it reads its destination address and the message is resent in the adequate output port (or multiple ports in the case of a multicast or broadcast message) - this behavior has to be programmed. The code will rely on the OMNeT++ simulation class library.

2.4.2 OMNeT++ adequate functionalities

- Like NS2, OMNeT++ is modular and allows the implementation of new modules using C++, which would allow for the particular implementation of the PLC network devices behavior as well as easy swapping of different protocol layers.
- Provides some graphical tools for aiding on the design and analysis of the simulation, which allows for faster and more pleasant mean of designing simulation scenarios and analysing simulation results.
- Using text format configuration files allows for faster changing of simulation parameters even by non-authors of the simulation modules, and requires no recompilation of the sources.
- Documentation is accurate and allows the developer to concentrate more on the simulation rather than the simulation tool.
- It has a very smooth learning curve, and in just a few hours the developer is able to start building its own simulations.

- It is very easy to embed third party C or C++ code, as well as to use precompiled modules written in other languages, or from whom there is no source code available – for this the user most program the interaction (communication) with the foreign module.

2.4.3 OMNeT++ inadequate functionalities

- Visualization tools are very simple and do not allow for backward playing of simulations, neither correctly shows simultaneous messages – instead a message at a time is shown.

2.5 Chosen simulation Tool

The chosen simulation tool was OMNeT++. This choice was obvious as it was the only simulation tool that shown potential to integrate third party simulation modules of different kinds. This is particularly important because a PLC physical layer emulator had already been implemented (see next chapter) by the iAd research and development company, and needed to be integrated with the simulator.

Beside that, it is very easy to learn, and the provided information on building new modules and its resources on that purpose (continuous or discrete event processing strategy engines, module links and gates, specialized message classes, random number generators, etc).

3 Simulator

3.1 Basic Network Layer Emulator

In order to develop and test the Transport Layer functionalities a Network Layer implementing all the services (even in a very basic way) was required. Although a third party Network Layer Simulator has been developed (see chapter 3.2.2), it was only available later on the Transport Layer development. To aid on the first stages of development, a Basic Network Layer Emulator was developed. [**Error! Reference source not found.**]

This Basic Network Layer Emulator has three versions, being the second version an evolution (with some improvements and added functionalities) of the first, and being the third an evolution of the second. Their capabilities were:

- **First version**
  - was able to logically interconnect several Transport Layers in a tree-like structure (with an Access Point on the root, Bridge Network Layers at mid-level, and Node Transport Layers in leafs).
  - supports the concept of multiple logical channels - some logical channels may be "disabled", i.e. they count for delay purposes but can not used for data transmission.
  - each Master-Slave connection has a minimum delay of one time slot and a maximum delay of 3 time slots (emulation of repeaters). If this is a request-response situation then the delay to the response is the same as the delay to the request. This delay is randomized.

- data delivered by to different channels in sequence may be delivered to destination out of order due to dynamic change in delays for each logical channels.
- each network layer has 2 priority queues for each active logical channel, and each queue may store up to 4 fragments.
- each channel has an error probability - this does not change the reserved delays. In the request-response situation, there are 3 possible outcomes: request lost, request delivered but no response received, request and response delivered. In the unicast send situation there are 2 possible outcomes: request lost, request delivered.
- in the master side when a unicast message is issued the Network Layer transmit queue data is removed even if the request is later lost due to the error probability function.
- in the slave side a response can only be issued after a confirmed request from the Master.
- in the master a slave polling system at periodic intervals issues a confirmed request to each slave in order to retrieve data that the slave may have to send.

- **Second Version**
  - support of multiple independent network units in Slave and Masters. They are unrelated since different units will use different non-overlapping logical channels. Each connexion from a Master or Slave has its own unit, allowing the creation of simple routing tables at the Transport Layer.
  - provides information about each link status like "error rate" with values from 0 (perfect link) to 100% (link disconnected) to the Transport Layer.
- **Third version**
  - support of Slave Login/Logoff procedure and Slave table update functions (at the Master).

This basic simulator of the Network Layer allowed the earlier development and test of the Transport Layer functionalities. However when a third party Network Layer Simulator was available, this basic simulator became obsolete. In next chapter are described third party simulation modules that were used in the simulation.

3.2 Third Party Simulation Modules

3.2.1 Physical Layer Emulator

Layer 2 and above simulating tools like OMNeT++ are often used for development and verification of network protocols. But the existing channel models in those simulators do not cover the specific behaviour of communication on power-line. Especially if several transmitters send in one network at the same time and frequency range the collision and useful superposition (e.g. single frequency network – actually the superposition of two equal signals can be benign and boost the readability instead of making the information unreadable) are not handled. To allow for the simulation of the REMPLI network a Physical Layer Emulator was developed by the iAd group.

But it is very hard and complex to emulate the behaviour of the Power Line Communication system. The Physical Layer Emulator emulates the behaviour of the channel and the physical layer of the PLC system, including channel encoding and synchronisation, and calculates the estimated response of the physical layer to the real signals on the power line.

Due to the high efforts on calculations for the real signals, this emulator is very time consuming, only allowing for the analysis of few transfer blocks and is therefore not suited for the simulation of a protocol. Based on this detailed simulation and the field results already available certain figures can be determined, which were the basis for a simpler version of this emulator that was used in the protocol simulator embed into an OMNeT++ module.

This simpler version abstracts the effects of the physical layer response to the electrical signals, enabling a lot more efficient protocol simulation. It also permits repeating the same simulating conditions for different simulation configurations allowing direct cause-effect comparison between them.

This emulator can be used to simulate a time slot for a Power Line Network for built-in scenarios close in design to most real ones:

- Logical ring of 10 devices;
- Logical ring of 100 devices;
- Logical open ring of 100 devices;
- "Random" area of 20 devices;
- "Random" area of 100 devices;
- "Random" area of 200 devices.

The scenario that implements a logical ring of 10 devices was used in the simulation. In the simulation structure (which is described in chapter 5.3) the maximum number of devices at one segment is 5, so the chosen PL Emulator scenario is enough.

The emulator is written in C language, and was easy to embed in an OMNeT++ module. Its functionalities are available through the use of function calls. In order to simulate a time slot, the messages sent by all devices within that time slot have to be scheduled in the emulator using the emulator function EmuTC_Phy_Tx(). Then the simulator has to call the emulator function Emulate() to calculate one time slot (i.e. to calculate which device receives which message and with which CRC status). Finally in the next time slot the simulator calls the emulator function EmuRC_Phy_Rx() to receive each messages scheduled in the previous time slot, and sends them to the calculated destinations.

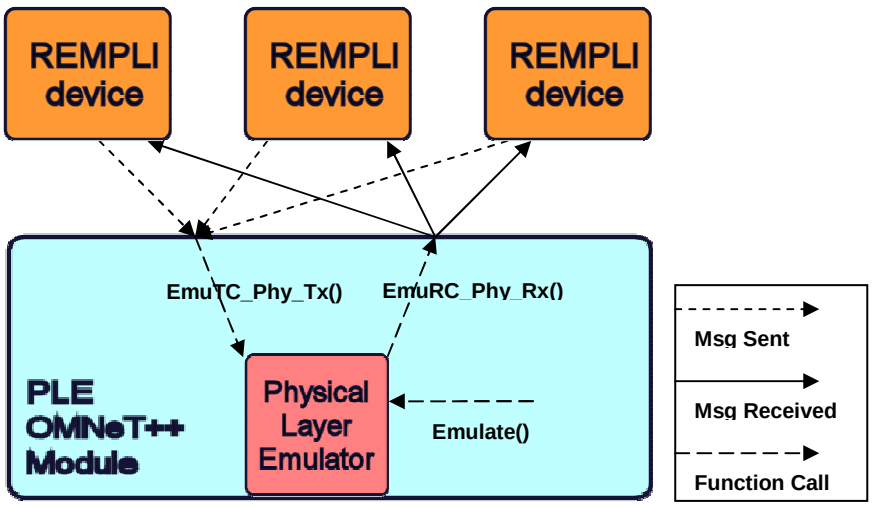

Figure 1 – Physical Layer Emulator Integration

3.2.2 Network Layer Simulator

Latter on the development of the simulation a more complete simulator for the REMPLI Network Layer was made available by the Loria institute (France). This new simulator replaced the Basic Network Layer Emulator that was previously developed. This simulator wraps the Physical Layer Emulator, and at its time is wrapped by the "PLC Network" OMNeT++ module. From the other modules was removed the Basic Network Layer Module (master or slave).

3.3 Structure

The simulation structure must mimic real Power Line Communication system implementation structures. In real implementations the PLC network may be compound by just a medium-voltage segment (see Figure 2), or also with a low-voltage segment (see Figure 3). The medium-voltage clients are mostly factories where its machinery has medium-voltage requirements.

3.3.1 Medium-voltage Simulation Structure Example

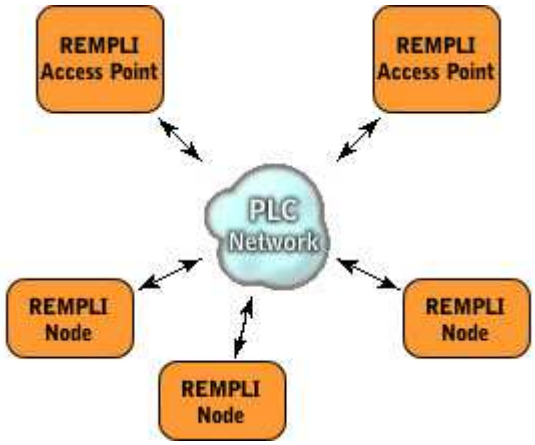

Figure 2 – Medium-voltage Simulation Structure example

In this scenario there are two Access Point devices at the power station enabling application communication via the PLC network. There are three Node devices installed at three clients that gather metering information that is periodically asked by Access Point applications. The Access Points represent two different power-line network interface points that could exist in different facilities, and the Nodes represent medium-voltage clients or transformation stations (stations that transform power from medium-voltage segments outputting low-voltage current on low-voltage segments).

3.3.2 Medium/Low-voltage Simulation Structure Example

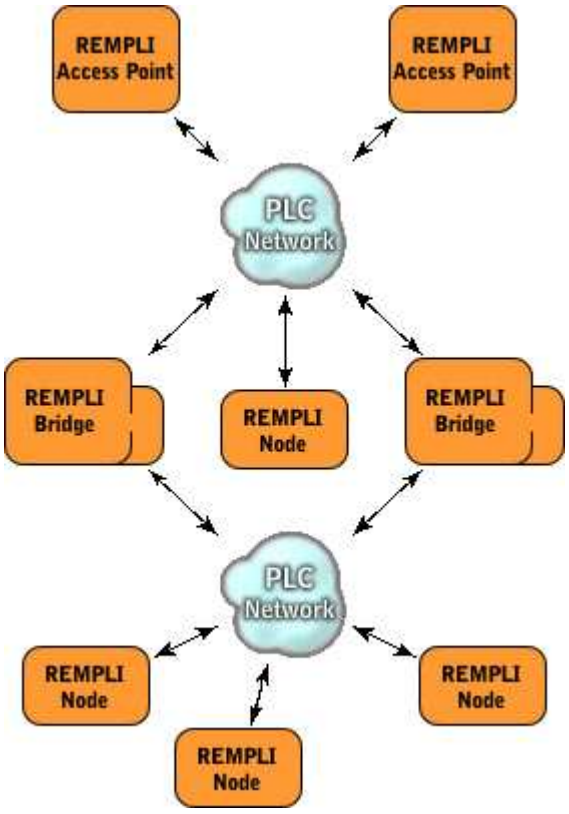

Figure 3 – Both Medium-voltage and Low-voltage Simulation Structure

In this example scenario there are two Access Point devices at the power station enabling application communication via the PLC network. There are two transforming stations with Bridge devices enabling the communication between the two different segments. There is just one Node device at the medium-voltage segment. At the low-voltage segment there are three Nodes installed at three clients gathering metering information that is periodically asked by Access Point applications. The Bridges also act as Nodes and can also provide data to Access Point applications.

3.3.3 PLC Network Module Structure

Due to the preliminary use of a Simple Network Layer module and the later usage of a third party Network Layer Simulator two different versions were implemented (as described earlier in this thesis). The most important difference is that in the first version the "PLC Network" module only embedded the "PLC Emulator", but in the second version it embeds the "Network Layer Simulator" that at is turn embeds the "PLC Emulator".

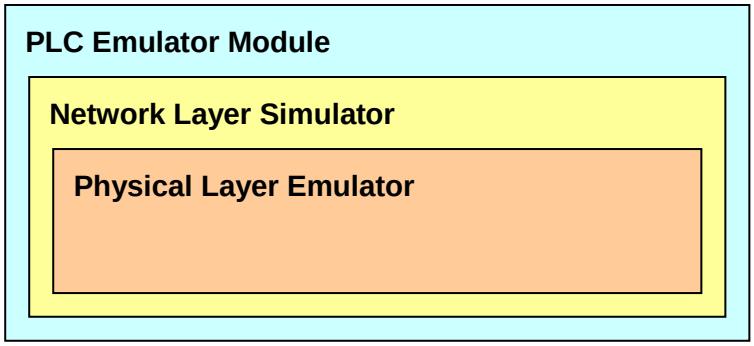

Figure 4 – PLC Network Module Structure

3.3.4 Access Point Module Structure

The first implementation included the Master version of the Simple Network Layer module (called "Master N.L." – see Figure 5), but in the second version that module was removed and the Transport Layer communicates with the Network Layer Simulator at the outside (via the "PLC Emulator" module). The other modules are the "Access Point Driver" which simulates the Access Point Driver, and the "Access Point T. L." which simulates the Access Point Transport Layer.

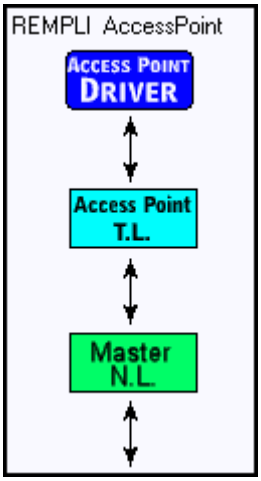

Figure 5 – First Access Point Structure

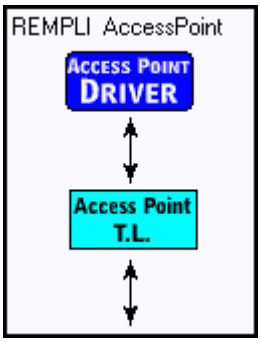

Figure 6 – Final Access Point Structure

3.3.5 Bridge Module Structure

As well as there are two versions of the Access Point, also two versions of the Bridge were implemented – with and without the Simple Network Layer Modules. The Bridge has to connect to two different Network Layers – a Slave Network Layer at the medium-voltage segment and a Master Network Layer at the low-voltage segment.

The other modules are the "Bridge T. L." which simulates the Bridge Transport Layer connecting the two different Network Layers, and the "Node Driver" which allows the Bridge to act as a Node providing services or data to the Access Point applications.

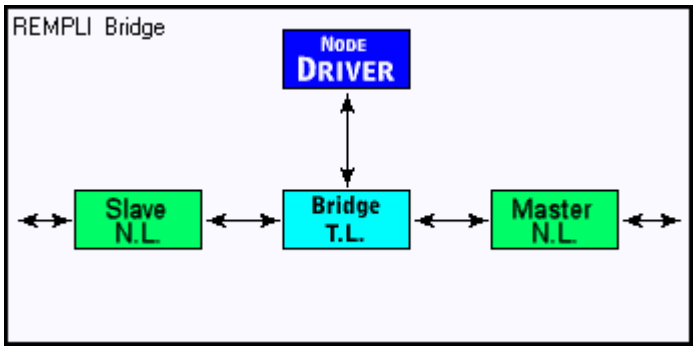

Figure 7 – First Bridge Structure

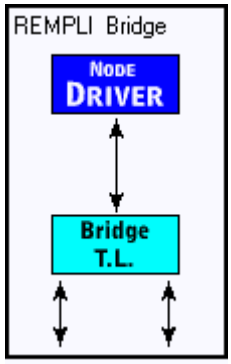

Figure 8 – Final Bridge Structure

3.3.6 Node Module Structure

There are also two versions of the Node – with and without the Simple Network Layer Module. There is also the "Node T. L." which simulates the Node Transport Layer, and the "Node Driver" (like the Bridge) which allows providing services or data to the Access Point applications.

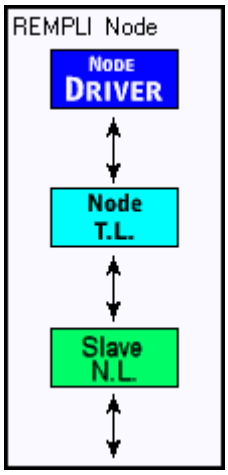

Figure 9 – First Node Structure

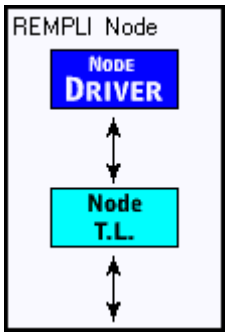

Figure 10 – Final Node Structure

3.3.7 Transport Layer Module Structure

The Transport Layer structure is equal in all of the three devices (Access Point, Bridge and Node). The only difference is the outer connections as the Bridge Transport Layer has connections to two different Network Layers while the others have only one.

The topmost module is the "RCI Manager" (REMPLI Communication Interface Manager) that receives requests from the drivers and sends them to "Queue Manager" or "Transmit Route Manager" modules accordingly (if it is a request for slave device's logins it is sent to TRM, and if it is a request for a Node service it is sent to QM). It also delivers the responses from those two modules to the drivers.

The "Queue Manager" module receives driver requests from "RCI Manager" and delivers them the responses and alarms. For every driver request it sends "Transmit Route Manager" new queue data and asks for routing information. When it receives an order to serve a queue a fragment from that queue is sent to the "Network Layer Interface".

The "Transmit Route Manager" module is responsible for the management of routes and for the queue fragment scheduling. At the master side it receives slave login information from "Network Layer Interface" that is used to create the routing tables. It also processes requests for slave information either from "Queue Manager" and drivers (via "RCI Manager").

The "Network Layer Interface" module is the interface between the two main modules and the Network Layer. It bridges messages between them and the Network Layer. The messages coming from the Network Layer side are analysed in order to be delivered to the correct module.

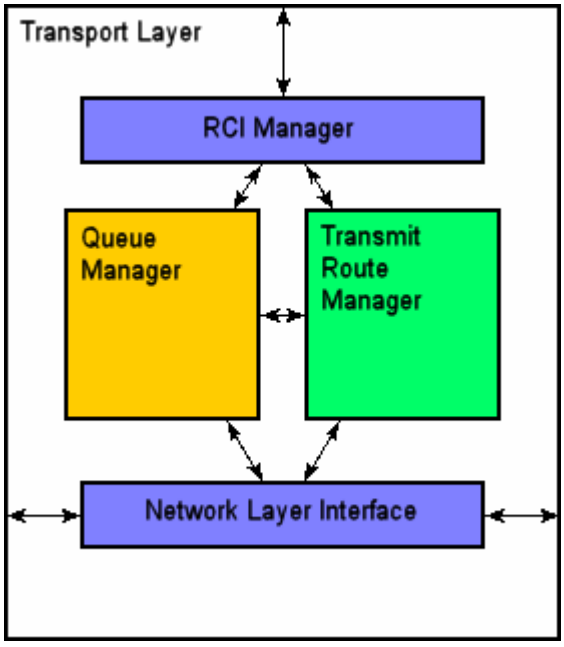

Figure 11 – Transport Layer Structure

3.4 Modules

3.4.1 Access Point Driver

In the simulator the Access Point Driver module does not behave as a real one. In a real device the driver would receive requests from extern applications and translate them into protocol requests. In the simulation there are no applications making requests. Instead the Access Point Driver module emulates application requests by making requests on its own. This is not a problem because the simulation objective is just to analyse the routing protocols on PLC Networks, which are not affected by the way the requests are being made.

This module was used to simulate the system with confirmed requests, unconfirmed unicast requests, unconfirmed multicast requests and status requests. Each of those request types (except status requests) foresees two headers with different size for PDU length (one allowing for a maximum PDU length of 511 bytes, and the other allowing for a maximum of 16777216). Therefore it is convenient to compare results of those request types at the boundary point and analyse if this particular optimization of having a special smaller header for small PDUs has the desired impact on network performance.

3.4.2 Node Driver

Like "Access Point Driver" this module emulates application requests instead of translating real application requests. It was used to simulate alarm requests as they are the only requests that can be issued at the Node Driver. The alarm service will mostly be used for sending very small PDUs so it is very important to get simulation results for such a scenario (e.g. alarm requests with length of 20 bytes). Due to the alarm service foreseeing two slightly different headers (one allowing for a maximum PDU length of 511 bytes, and the other allowing for a maximum of 16777216) it is convenient to compare the results of sending alarm requests at the boundary point, to check if the optimization has the desired impact on alarm propagation and confirmation times.

3.4.3 RCI Manager

The RCI Manager module interfaces the Transport Layer with the drivers. As was previously described it processes the incoming messages from each other module and delivers them to the correct destination. But there can be many drivers connected to the RCI Manager, so to correctly deliver notification and responses to driver requests it is necessary to store information about what driver sent what request. This is achieved by storing that request information among its IpcTransactionID. Every response or notification with that IpcTransactionID could then be delivered to the corresponding Driver DeMux.

3.4.4 Queue Manager

The Queue Manager is along with Transmit Route Manager one of the most important Transport Layer modules. It is responsible for the managing of the message queues, including all the information related to them as well as fragment serving and confirming.

At the Access Point it was used to:

- Create transmit queues (outgoing queues) and receiving queues (incoming queues);
- Ask TRM for route information to destination devices;
- Create fragments from PDU data according to the available message size at the communication medium;

- Send fragments from queue as instructed by TRM;
- Manage queue information and its fragment status;
- Notify TRM of acknowledges and errors on outgoing fragments;
- Reassemble incoming fragments and deliver responses and notifications to the Drivers;
- Send confirmation messages acknowledging the reception of fragments on confirmed services;

At the Node side it was also used to:

- Generate and attach an special IpcTransactionID to a confirmed request (request with response service) when delivering it to the Node Driver (used to pair the response with the request);

At the Bridge side it also:

- Reassembles incoming fragments data from each communication segment and reconstructs new fragments according to the (most likely) different message size at the outgoing communication segment;

3.4.5 Transmit Route Manager

The Transmit Route Manager is the other considerably important Transport Layer module. It is responsible for the slave login mechanism and for the management of the information about routes between all devices. It is also responsible for taking the decision of what Queue Manager queues to serve at each time.

This module was used to:

- Create and manage slave login lists;
- Maintain link status and routing information;
- Choose the best available route for sending each request;
- Keep information of available time slots;
- Choose which queue to serve at each time;
- Take into account queue priorities when choosing a queue to be served;
- Collect status bits from slaves;
- At the Bridge: forward slave login lists and route information to the TRMs at Access Points.

3.4.6 NL Interface

The NL Interface module interfaces the Transport Layer with a Master Network Layer (at the Access Point) a Slave Network Layer (at the Node) or both (at the Bridge). It processes the incoming from the Queue Manager and the Transmit Route Manager, saves queue information and adds a NetTransID (network transaction ID), and sends them to the adequate Network Layer. In the other direction the messages and notifications from the Network Layers are processed and sent to the respective module (Queue Manager or Transmit Route Manager). The previously saved information associated to the incoming NetTransID is added to the notifications so QM and TRM know at which queue they correspond. But when receiving requests from the Network Layer no data is append because there is no stored data.